EPISODE 713

[INTRODUCTION]

[00:00:00] JM: TLA+ is a formal specification language. TLA+ is used to design model and verify concurrent systems. TLA+ allows the user to describe the system formally with simple precise mathematics. TLA+ was designed by Leslie Lamport, a computer scientist and Turing Award winner. Leslie joins the show to talk about the purpose of TLA+.

Since its creation in 1999, TLA+ has been used to discover bugs in systems such as Amazon S3, DynamoDB, Xbox and Azure CosmosDB. TLA stands for temporal logic of actions, a logical system that can be used to describe the behaviors of concurrent systems. This podcast is meant as a brief introduction of TLA+. To go deeper, check out the TLA+ website and the TLA+ video course, both of which are linked to in the show notes of this episode.

As a side note, the videos in the TLA+ video course are highly entertaining because of Leslie's dry, unpredictable sense of humor. I recommend them, and I hope you enjoy today's episode.

[INTERVIEW]

[00:01:19] JM: Leslie Lamport, you are the creator of TLA+. Welcome to Software Engineering Daily.

[00:01:24] LL: Thank you.

[00:01:24] JM: What is TLA+?

[00:01:26] LL: TLA+ is a language for writing high level models of digital systems. A real digital system is something that operates with transistors and sending signals over the wires doing all sorts of things and is described by partial differential equations, etc., etc. But that's not the way we want to think of our digital systems. A digital system is one that we can think of or is designed so we can think of as operating in a sequence of distinct steps, and a model of that system is one that models its real execution as a sequence of distinct steps.

[00:02:12] JM: Why would we want to model these real-world systems before we actually start to create them?

[00:02:18] LL: What is creating them in, you'd probably want to figure out what a computer is supposed to do before you start wiring up transistors. Similarly, you'd probably like to figure out what million line program is supposed to do before you start writing code.

[00:02:36] JM: What's an example of the system that we would want to model before we start working on it? Maybe you could give a real-world application that somebody has modeled in TLA+ and made good use of it.

[00:02:50] LL: Well, first of all, it depends what you mean by a model. Everyone who writes a program has some kind of model in his head, some idea of what it's supposed to do. It doesn't start with a blank screen and say, "I'm just going to write a bunch of code and see what happens."

The question is not whether you should be modeling systems. Is the question of how much effort you should expand in making the model before you start building the system, and TLA+ is designed for systems for which you really want to make sure that you're getting them right. So you want to be able to check your high-level models before you go any further.

[00:03:40] JM: You've given a motivation for why we would want to model a system. A TLA+ spec is a model for a program. It's not the program itself. What is the difference between this spec, this model and an actual program?

[00:03:58] LL: Perhaps a factor of hundred in size and – Well, the reason you need to expand your model by factor of a thousand or so is that if you build a model at the level at which you really want to think about these things without getting unnecessary details to complicate things, you're not going to be able to generate code from it.

[00:04:25] JM: What's useful about a TLA+ spec is that it can be evaluated, and by evaluating this spec, we can find logical gaps in our model. If we find these logical gaps before bringing the

model into further implementation, we might be able to avert a lot of wasted time or perhaps some critical error. Can you explain what it means to evaluate a spec?

[00:04:53] LL: Well, the ideal tool to do that or the ideal way to do that would be to write a machine check mathematical proof, which would mean that you're actually proving that every possible execution of the system, or more precisely, every possible behavior allowed by the model is correct. Unfortunately, writing machine check proofs is very hard and expensive and you would only really want to do it when it's really, really life-critical systems.

The tool that's mainly used by engineers is the model checker. Now, a model checker in principle evaluates or finds all possible behaviors of the model and checks them, but now there's little language problem because there are two meanings to the word model that I'm about to use. The model checker, that model, is actually a model of the TLA+ model.

So let me start calling the – Instead of calling the TLA+, what you write in TLA+ a model, let me call it a spec specification. So what you do with a model checker is you make a model of the specification. For example, your specification might be of a system with any number and of processes and model of it that the model checker looks at might say to check in with three processes. Even with a fixed number of processes, many models will allow an infinite number of possible behaviors. So you'll use some kind of method to restrict the size of the model of the number of behaviors so that it's finite and the model checker can generate all those behaviors.

For example, you might say just consider behaviors in with which of the lengths of cues is less than some number. Model checking turns out to be amazingly effective at finding subtle errors that people are very easily miss and very – It's very hard for people to find and even on very small models. So engineers find that pretty satisfactory to the extent that engineers are willing to accept a TLA+ spec as correct with models that would naïvely seem to be much too small. For example, Microsoft and Amazon build systems containing hundreds or thousands of computers and they might be able to check their algorithm with three or four computers. But that gives the engineers the confidence that they need in order to go ahead and build a system, which has to be very reliable.

[00:07:59] JM: We'll ease into a discussion of the practical applications of TLA+. If you imagine a system like a distributed database, or a heart rate monitor, or an airplane, these are systems that contain lots of different moving parts, and in order to model them with something like TLA+, we might choose to scope our model to the critical parts of a digital system. How can we find the critical parts of our system so that we can focus on modeling those before we start the implementation process?

[00:08:40] LL: Well, that's what engineering is all about. Mathematicians would have a real easy time learning TLA+, much easier for them than for an engineer. But having learned TLA+, a mathematician is not going to be able to design a distributed database that we would want to use. There's a lot of engineering experience that needs to go into that. I can't give you a general prescription of how you do that, but an engineer, good engineer, will know what are the parts that are hard to get right and what are the parts that just a bunch of details where they will have other tools in order to check those mistakes.

For example, if you're looking at a distributed database – I mean, people have been building databases for years or decades, but what's new with the cloud is that those databases are being distributed over thousands of computers. So the engineers aren't worried about how they actually execute those database commands. How do you do this over a piece of data or something? But what the problem is –Well, there might be two major problems. One is making sure that if you've got multiple copies of the database, they're consistent, and another problem might be, well, if you've got the – You want to try to make sure that the data is in the place where it's going to be used, and had you managed to move the data while it's being used? Those are very subtle concurrency problems, and TLA+ is great at helping you get the concurrency aspect of things right. It's not particularly good in making sure that you're executing this particular operation on this particular kind of data correctly.

[00:10:42] JM: A specific example of an application is that TLA+ is used by AWS S3, the Amazon S3 storage system, and I know they sent you an email at some point about how useful the TLA+ language was to them. In your conversations with the industrial appliers of TLA +2 to, for example, cloud services like S3 or something in Azure, how have those usages of TLA+ translated to material gains in the software development process?

[00:11:23] LL: I'm not sure how to answer that. I can't give you something that says, “Oh, they've saved X thousand dollars or X million dollars by using TLA+.” I can say that on several of their systems, they have found bugs that could have caused loss of user’s data. Those bugs, they would not have found without using TLA+. Amazon does not want their services to lose customer data. But I don’t know how you would put a dollar figure on how much not losing customer data was worth to them, but they certainly believe it's worth enough that they make TLA+ a standard part of their development process when they have that kind of critical problem.

[00:12:17] JM: Yes, they’ve said in – To summarize, they’ve said that it forces the authors of systems to think more clearly and the tools can be used to check their errors, which you can assume leads to saved time and saved dollars. Before we get to talking about the specifics of building models in TLA+ or building specs, has TLA+ plus been useful for you for solving your own problems and modeling algorithms that you’re working on?

[00:12:49] LL: Listeners are probably most interested and if they're going to be writing programs, is this going to help them? I don't know too much programming these days. My programming days has pretty much ended about 40 years ago or so, but for the past 10 or so, 15 years, I've been writing an occasional piece of code, mostly as just for implementing small things inside of TLA+ tools. In those 10 or 15 years, I've probably used TLA+ itself may be a half-dozen times. That is I will have something that is I feel is complicated enough and difficult enough to get right that I want to have this tool to check it.

Now, I can't say that TLA+ is the best possible tool for that, because it turns out that I actually haven’t been doing any concurrent programming in those past 15 years. Since it's a tool that I know it's better to use that than a tool that might be better for this specific application but then I’d have to learn. But more than that, I would say my experience, and it’s hard for me to say do I do things the way I do because I've been using TLA+ or that I built TLA+ to support the way I want to do things? But the basic idea of thinking in terms of a model and of writing down what that model is supposed to do is useful in everything, all the programming that I do.

So if you look at any piece of code I've written, if it's not really trivial, you’ll find that there are comments explaining what the code is supposed to do that are much longer than the code itself, because I believe that it's really, really important to think before you code. Coding should be

trivial. If you find that you're having trouble figuring out how to code something, it's probably because you don't understand exactly what it should do, or exactly how it should do it. To think about those things, you want to be thinking above the code level.

To give you a trivial example, your problem is finding a greatest common divisor of two numbers. Agin, everybody who study computer science knows you do that with something like Euclid's algorithm. But imagine if you didn't know Euclid's algorithm and we're trying to write a greatest common divisor program and you were thinking in terms of code, in terms of while loops and stuff like that. How would you possibly discover Euclid's algorithm or discover a good algorithm for finding a greatest common divisor. If you are thinking in those low-level terms, Euclid's algorithm is a – Finding of the greatest common divisor is a mathematical problem. So you should be thinking mathematically, not in terms of code.

Tony Hoare said something years ago, he said inside every big program is a little program trying to get out. I interpret that as meaning that inside every big program, there's a small algorithm trying to get out. Probably not an algorithm that you would've learned in an algorithms class, but the difference is not that it's something inherently different, it's just that it's only an algorithm good for the thing, particular thing this program is doing and other people aren't going to be interested. So you're not going to find it in an algorithm textbook. So you need to be thinking at the algorithm level rather than at the code level.

[00:16:40] JM: I was going to say, when I took some of my computer science theory classes, there was a widespread use of state machines, and TLA+ is – You described it, or I think somewhere on the TLA+ website, it is described as a state machine language, and state machines are a simple and powerful abstraction for modeling systems in a clean way. Is TLA+ – Would you describe it as a state machine language?

[00:17:10] LL: If you call it a state machine language, that's going to confuse people who have studied computer science, because the state machines that they've seen have probably been finite state machines. One state machine that they have seen that isn't finite state is a Turing machine, and that's why Turing machines can do most anything and finite state machines can't do much. At least they're just not very good for modeling complex systems.

The kind of state machine that I'm talking about is if you start thinking about a Turing machine, then you're heading in the right direction. Well, a Turing machine is a particular state machine or the kind of state machine that's made for certain purpose. What I mean by a state machine is something that you can describe what it does, but as generating a set of behaviors, where behavior is a sequence of states, and all a state machine consists of is a rule for telling you what the possible initial states are and a rule for telling you how to get from one state to possible next states. That's all a state machine is.

I think pretty much all the models that people have in their heads about how digital systems work, like how programs work, are state machines, although they're not described precisely. You think of a program as taking a series of steps, and in each step it changes the state. It changes the values of the variables. It moves control to the next statement in the program or something like that. But what TLA+ does is allows you to write state machines in a precise mathematical language. Being mathematical means that you can basically express anything you can express mathematically in TLA+, which makes the language more expressive, which makes it better at writing high-level models. It also makes it correspondingly harder to generate code from it.

[00:19:17] JM: If we're using TLA+ to define the states of our program, what would be an example of defining the initial state of a program and defining the formula for how the program advances from one state to a different state?

[00:19:37] LL: It's very hard to do that on the radio.

[00:19:41] JM: Understood.

[00:19:43] LL: Let me give you the most trivial example I can think of. This program that has a single variable X and it starts with X = 0 and it keeps incrementing X by 1. In the first state, X = 0. The second state, X = 1, and the third state, X = 2 and so on forever. Roughly speaking, you would specify that by saying the initial state is described by saying X = 0 and the next state relation, the thing that tells you how to go from one state to the next is I write it as X prime = X +1, where X prime means the value of X in the next state and X means its value in the current state. So it just says X prime, the value in the next state, equals X the value in the current state

+ 1, and that's all there is. Well, a very trivial example, but as you can see it also has a very trivial description as a state machine.

[00:20:38] JM: Absolutely. Just to ease any anxieties about describing this over the radio, definitely are thousands of people out there listening who are vividly entertained by hearing about a simple state machine over the radio. It's just that it's a fairly new format, computer science talk radio, but it is widely appreciated by people on their commute. So you are not falling on deaf ear just to provide you some solace.

There is a concept in computer science theory, or maybe it's a mathematical theory, called the invariant. In defining a TLA+ spec, you need to be aware of this idea of inductive invariants. What is inductive invariants?

[00:21:29] LL: Okay, first let's start with what an invariant is. An invariant of an algorithm or of a specification is a statement about states. It's a predicate on states that's true for every possible state that the algorithm where the specification can reach. Now, invariants are crucially important, because the reason – What a program or what a specification does next doesn't depend on what it did in the past. It just depends on what's in its current state. That's true for specifications. That's true for computers. It's true for bundles of transistors. Everything or the next thing that it does depends only on what's in its current state.

If you understand the algorithm, that is understanding what makes it behave correctly, means understanding the invariant that at each stage, at each step, guarantees that in the next step, it's going to do the right thing. So that's what an invariant is. Now, an inductive invariant is a restricted – It's a special case of an invariant that is used to – If you want to write a proof that something is invariant. I don't think it's worth telling you exactly what it means, but think of it as a proof of invariants is an induction proof. You basically want to show that if it's true in any particular – If it's true for any – Well, it's true initially in the first state and if it's true in the end step of the algorithm, then it's going to be true in the end plus first, and then by induction, if that's true for all end, we know it's true for all states.

Well, if you've ever done proofs by induction, you know that you sometimes in order to write an induction proof, you have to strengthen the thing you're trying to prove. So the induction works.

So instead of proving the thing that you want to proof, you have to prove something stronger just to make the induction work. That's what an inductive invariant. It's something that's stronger than the invariant you're interested in, but that you need in order to make the induction step work.

[00:23:54] JM: When proving things about a distributed system, we are often trying to prove the maintenance of liveness, which is that the system will continue to execute productively. We also want to emphasize that safety is being held true. Meaning, maybe wanted to define these things, but my understanding of safety is that you're not going to have something – Maybe in a practical applied sense, you're not going to lose data, for example.

In TLA+, how our safety and liveness conditions represented? What is safety and liveness in your definition?

[00:24:35] LL: Well, I first gave an informal definition, and since then there's been a formal definition given. So these are precise terms and it's not just my definition. But, informally, a safety property is something that says what the system or what the specification is allowed to do, and liveness property is a property that says something that the system must do.

For example, if you've studied program correctness of sequential programs, you know that's always split into two parts. There's partial correctness, which says that if the program stops, then it's going to stop with the right answer, or another way of saying that is it's only allowed to stop if it has the right output. There's termination, which says eventually the program must stop. Well, partial correctness is a safety property and terminations is liveness property, and when reasoning about systems, you really want to keep those two things separated.

What I told you about a state machine is it's actually half the story. When I said that its initial predicate at the next state relation, that's describing safety. You need to add something to describe liveness, and the best way of doing that is in terms of what's called fairness. I think it's best if I don't try to go into explaining exactly what fairness means. But an example of a fairness condition, you would want to add for that little program that just keeps incrementing X by 1 is to say that it never stops and you would do that by saying there's a fairness condition on the next

state relation. But once you get into concurrency, there are a lot more types of the liveness properties that want to express, but generally they're all types of fairness.

Now, the TL in TLA+ stands for temporal logic. I don't usually tell people that, because I don't want to scare them. Temporal logic was introduced into computer science in 1977 by a Amir Pnueli, and when I saw it and tried playing with it a bit, it became clear to me that that was exactly the right thing you wanted to use to talk about liveness. So that's why I started using it.

But it turns out that kind of use of temporal logic, it's not useful for safety except when you doing the reasoning it's safety – When you're using TLA+, the safety part, which is basically the initial predicate in the next state relation, it's sort of wrapped into a temporal logic formula, but you don't even have to know that. The tools don't care whether you give it that formula or just say here's the initial predicate in the next state relation.

But to specify safety, you have to write a temporal logic formula. It's a pretty simple formula, but it's something else that – something that you need. But with safety, that temporal logic and the TLA+, the temporal logic part of it, the TLA part of it, is really great.

[00:27:59] JM: The way I look at this interview is to give people a bit of a teaser for what TLA+ is and if it sounds like it might be useful to encourage them to go to the website and check it out and watch, by the way, your amazing videos. You have some really entertaining videos on this. So I really do look at this conversation as fairly introductory, and I understand if we can't cover all of the aspects of TLA+. I do believe some people out there will find out about it. So I think this will be a useful interview. Why should the listeners check out TLA+?

[00:28:35] LL: Well, let me give you one anecdote, and remember this is just one anecdote. There's a European space agency spacecraft that landed on a comet a few years ago, and inside of that space there was a real-time operating system that was controlling some of the [inaudible 00:28:58] instruments, it's called Virtuoso. People who built Virtuoso decided to build the next version, but they decided they want to base through it using formal methods and they decided on doing it in TLA+.

I found out about it just because they wrote a book about it. So I sent an email to the head of the team and asked sort of what their experience was with TLA+. What he told me is that as a result of using TLA+, the new system used 1/10th of the code that the original system had used. The reason was that by being able to think about what they were doing above the code level and think mathematically, they were able to come to a much cleaner design than they had been able to do before.

I'm not going to guarantee that anybody who's – If they use TLA+, they're going to have 10 times less code in their next project. But that should make them start thinking, that maybe they should be doing something differently from what they're doing, because there's so much emphasis on coding, but no better coding is not going to reduce your program size by a factor of 10. That's only going to happen through better thinking. That comes by thinking above the code level.

So what I will tell your listeners is that I don't know if TLA+ is going to help them in what they're doing, or it's going to help them write their next – Is going to help them with their next program in the sense that they will be able to write a TLA+ model and then checking it is going to help avoid errors and all the things that reasons that people at Amazon and Microsoft were building complex distributed systems use it. But what I can tell them is that if they learn TLA+ and get some experience trying, it's going to improve their thinking, and that is going to improve their coding.

[00:31:15] JM: What are some key ways in which mathematicians and programmers differ in their typical thought processes?

[00:31:24] LL: Mathematicians think mathematically, and programmers don't. They tend to think in terms of code.

[00:31:30] JM: Would you say the TLA+ encourages programmers to think more like mathematicians?

[00:31:35] LL: I would say to some extent it forces them to, because you're going to have to come to terms with the fact that what they're writing is a mathematical formula. Now I should

correct that. It encourages them to come to grips with it. Engineers, if they still talk of a TLA+ spec as code, and there's PlusCal, which is an entryway to TLA+ that looks like a little toy programming language, except it's infinitely more expressive, because the expressions that looks like a toy language, except the expressions can be any formula of mathematics. So it makes it enormously expressive.

But it looks enough like the programming language that they'll find easy entry. But that [inaudible 00:32:25] entry is at the same time going to be a drawback in trying to get the benefit of learning to think mathematically. So they're looking for a nice tool to try. They're actually a couple of websites that have examples or one has a little video course about using TLA+, and they're great and I encourage people to look at them. But I also encourage them afterwards to try to find out about using TLA+ in the raw.

A PlusCal algorithm gets translated into a TLA+ spec. So as you're using PlusCal, especially as you're trying to debug your algorithm, you wind up looking at the TLA+ translation, which by the way is it's not like translating a program into a machine language. It expands the size by maybe a factor of two or three, but the TLA+ translation is quite readable. You may wind up looking at it when you're debugging your PlusCal spec. So it is a nice gentle introduction for people who are frightened by mathematics, and many if not most programmers seem to be. But I really encourage them at some point to bite the bullet and start using TLA+ in the raw.

[00:33:49] JM: Your sequence of videos on YouTube are also really useful resources, and they have a great deal of subtle humor. They're pretty funny, to be honest. You're changing outfits throughout these YouTube videos. So it's kind of surreal in some ways, especially because this dry humor contrasts with kind of the seriousness and the cleanness of your reasoning. Do you see humor as serving a utility in presenting computer science concepts?

[00:34:31] LL: I don't know. It keeps me interested in and what I'm doing, if I can put some humor into it. I don't know if I should give this away, but there's actually a reason for those costume changes. They break the continuity, which means that if I need to fix some part of the video, I can just change that part of the video and I don't have to worry about being in the same costume. Is my hair the same length and all of that?

[00:35:01] JM: Now, is there some sacrifice there? Because then you have to be deliberately changing outfits in the first iteration of the video.

[00:35:10] LL: Yeah, but believe me, changing outfits is a very minor part of the work that goes into producing a video.

[00:35:19] JM: Especially because some of those are like putting on a clown nose or a hat. Okay. Well, Leslie Lamport, it's been really great having a chance to chat with you about TLA+, and I think we'll have motivated some people to check out the language more. Thank you for creating it. Thank you for your contributions to computer science, and thanks for coming on Software Engineering Daily.

[00:35:39] LL: Okay. Well, thank you. It's been a pleasure.

[END]